New Generation Computing, 10 (1992) 287-313
OHMSHA, LTD. and Springer-Verlag

Highly Parallel Execution of Production Systems: A Model, Algorithms and Architecture*

Kemal OFLAZER
Department of Computer Engineering and Information Sciences,
Bilkent University,
Bilkent, Ankara, 06533 Turkey.

Received 15 October 1990
Final manuscript received 28 October 1991

Abstract This paper presents a new parallel processing scheme called DYNAMIC-JOIN for OPS5-like production systems along with associated parallel algorithms, a parallel architecture and simulation results from a number of production systems. The main motivation behind DYNAMIC-JOIN is to reduce the variations in the processing time requirements and improve limited production level parallelism. For this, the model employs some redundancy that allows the processing of a production to be divided into units of small granularity each of which can be processed in parallel. As a consequence in addition to production level parallelism where a set of relevant productions are processed in parallel, a second level of parallelism can be exploited.

After a detailed description of the model proposed, the paper presents algorithms for processing productions with DYNAMIC-JOIN, along with a discussion of various issues and possible disadvantages. Subsequently, the paper presents a parallel processor architecture that can implement DYNAMIC-JOIN, along with simulation results from real production systems.

Keywords: Production Systems, Parallel Processing, Massively Parallel Architectures.

* This work was done as a part of the author's doctoral thesis at Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A. It was supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, or the United States Government.

§1 Introduction

This paper presents a new parallel processing method called DYNAMIC-JOIN, associated parallel algorithms and a parallel architecture along with simulation results, for OPS5-like production systems. The main motivation behind DYNAMIC-JOIN is to reduce the variations in the processing time requirements and hence improve limited production level parallelism. For this, the model employs some redundancy that allows the processing of a production to be divided into units of small granularity each of which can be processed in parallel. As a consequence, in addition to production level parallelism where a set of relevant productions are processed in parallel, a second level of parallelism can be exploited.

After an overview of parallelism in production systems, the paper first presents a detailed description of the DYNAMIC-JOIN data and processing model. Subsequently, the paper presents a parallel processor architecture for implementing implement DYNAMIC-JOIN. Results on the run-time behavior of the algorithm and the performance of the parallel processor obtained from a number of large production systems like R1, XSEL are presented.

§2 Production Systems

Production systems are general computational mechanisms that have been employed as a programming paradigm in artificial intelligence where computation proceeds by applying rules in a sequence determined by the data and/or goals. They have been a paradigm of choice for building a class of programs known as expert systems. A production system consists of a *set of rules* called *productions* that make up the *production memory* and a global database called the *working memory*. In general, a production is a statement of the form:

$$P\colon\ C_1 C_2 \ldots C_c \rightarrow A_1 A_2 \ldots A_a$$

where C_1 through C_c are conditions—called the left hand side or the antecedents and A_1 through A_a are actions—called the right hand side or the consequents. Conditions are partially specified patterns to be evaluated on the current state of the working memory. Actions are executed when all the conditions of a production are satisfied and the production is selected for firing. The actions of the firing rule modify the contents of the database, enabling other productions for execution. The *production system interpreter* is the underlying mechanism that determines which productions are satisfied with the current sate of the working memory and should be executed. In general, the interpreter for a production system executes productions in a *recognize-act* cycle. Since the OPS5 system[2] will be used throughout the paper, its recognize-act cycle will be outlined here. The OPS5 interpreter goes through the following phases in the recognize-act cycle:

- MATCH: The LHS conditions of the productions in PM are evaluated to determine which productions are satisfied with the current state of the working memory.
- CONFLICT RESOLUTION: One of the productions from the set of matching production—*conflict set*—is selected for execution.
- ACT: The actions specified in the RHS of the selected production are performed. The working memory actions (i.e., make, remove, and modify) modify the state of the working memory. Other actions may perform input/output or any other computation.

Forgy[1] has observed that production system interpreters typically spend more than 90% of their time in the match phase and hence any significant speed-up in the execution of production systems will result from exploiting and improving any parallelism in the match process.* Match essentially involves finding which of the productions in the production memory are satisfied with the WMEs in the working memory. Forgy[3] has noted the slow rate of change of the working memory and has suggested that saving match state across cycles saves a considerable amount of computation. Thus match becomes an incremental computation where only the changes to working memory are matched to the productions and to any state associated with productions accumulated during previous match cycles.

The state of a production is that portion of the current working memory that is relevant to the condition elements of that production, and some other auxiliary information regarding how the working memory elements are joined together to form partial or complete instantiations. Figure 1 presents a simple view of the computation model that is employed in an interpreter that keeps a state associated with each production. With this model, the match process has

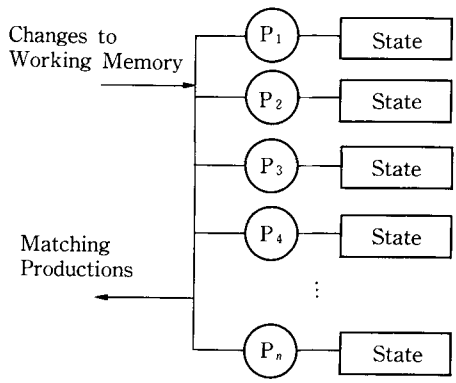

Fig. 1 Computation model with production states.

* However very recent progress in building special optimized (uniprocessor) compilers has been effective and has reduced this number down to around 50%[10] and this indicates that improvement efforts should start considering other phases of production system execution.

two phases: (1) **Selection** where productions whose states will be affected as a result of the action are determined.* (2) **Production State Update** where states of the productions that are selected by the selection are updated. If any of the productions are fully satisfied, they are inserted into the conflict set for conflict resolution.

2.1 Parallelism in Production Systems

Superficially, production systems appear to manifest a very high degree of parallelism. Productions, being independent of each other, can all be processed in parallel in response to change to the working memory. However, it is likely that as a result of a working memory action, only a small number of productions will be affected. One can intuitively speculate on this by noting that most production systems are composed of a large number of small modules, each of which work on some aspect of the problem. The productions in one module are designed to influence productions either in the same module or in other modules with which that module interacts. Hence when a production's action modifies the working memory by inserting or deleting a working memory element (WME), only a few out of the large number of productions need to be processed. It is this small subset of the productions that needs to be processed in order to the determine the contents of the conflict set, instead of the complete set of productions.

Exploiting production level parallelism involves processing the productions that are affected by a change to the working memory in parallel. Gupta's measurements[4] and our analyses indicate that for most of the production systems, on the average 20 to 30 productions need to be processed as a result of a change to the working memory although there are exceptions to this. However, the amount of work involved in processing each production may vary considerably. This reduces the parallelism at the production level since the time a match cycle takes is determined by the production which takes the longest time to be processed. So if one had a large number of processors—one to a production at the extreme—most of them would be idle either having no work to do, or waiting for some other processor to finish.

Gupta has investigated other levels of parallelism that can be extracted from the uniprocessor OPS5 RETE interpreter and has incorporated these sources of parallelism into an interpreter for shared memory multiprocessor systems.[4,6,5] Further lower level parallelism at a finer level of granularity is also potentially available. For example, Stolfo[14,16] presents a massively parallel machine, DADO, for parallel processing of production systems. Miranker[8,9] has devised a match algorithm for DADO which is dynamic variant of RETE. Similarly Hillyer and Shaw[7] present a scheme for using a massively parallel machine NON-VON to employ associative processing for exploiting lower level

* A production is affected by a working memory transaction, when the WME inserted or deleted matches at least one of the conditions of that production.

parallelism in pattern and variable matching operations of the RETE interpreter. Schreiner and Zimmermann[13] have reported a system which consists of a data-driven pipeline of special-purpose processing elements. They report rather high execution rates but the systems they simulate have a very small number of productions (around 10) which is not realistic. Perlin[12] has presented a mathematical framework for an incremental matching algorithm for determining the tuple instantiations of forward chaining production rules and it is claimed that the match operation can be performed in constant time provided sufficient number of processors are available. This formulation however relies substantially on the fact that the domains of all possible values for variables used in intercondition matching are known beforehand which is almost never possible.

§3 Processing the State of a Production

State processing is by far the dominant component in match. Naturally, there are a number of ways of how state processing can be implemented, each maintaining some intermediate information in some form or the other. For example, RETE[3] maintains only the individual condition element memories (α-memories) as WMEs are inserted or deleted until some WME matches the first condition element of the production. At this point, it starts joining them in a fixed sequence until no further joins are possible. Assuming R_1, ..., R_c denote the α-memories of the conditions of a production with c conditions, the information kept by RETE for a production is:*

- R_1, ..., R_c, as the condition element memories (α-memories.)
- $R_1 \otimes R_2$, $R_1 \otimes R_2 \otimes R_3$, ..., $R_1 \otimes R_2 \ldots \otimes R_j$ as the intermediate information, where either $|R_{j+1}| = 0$, hence no further joins can be performed, or $j = c$ (β-memories.)**

As WMEs are inserted to or deleted from one of the R_i the intermediate information is updated, along with the affected condition element memories. On the other hand, the TREAT algorithm,[8] keeps no intermediate join information but only the R_i. It recomputes the join whenever a new working memory element is inserted to one of the R_i and none of the others is empty. This gives TREAT the flexibility of dynamically ordering the intermediate joins so as to minimize the amount of intermediate computation. Figure 2 shows the state representations in TREAT and RETE for a production with $c = 3$ condition elements.

The state maintenance schemes of RETE and TREAT represent a rather conservative approach in that they try to reduce or minimize the total amount

* It is assumed that all the condition elements are positive.

** $\otimes$ denotes the join operation. For example, $R_1 \otimes R_2$ contains all pairs of working memory elements the first of which is in R_1 and the second in R_2 and are mutually compatible with respect to any variable references between the first and second condition elements.

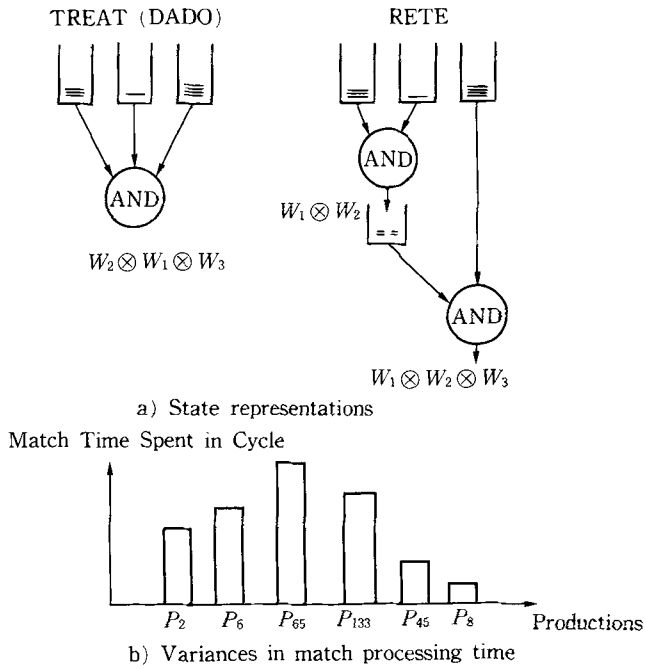

Fig. 2 State representations in RETE and TREAT and the resulting variances.

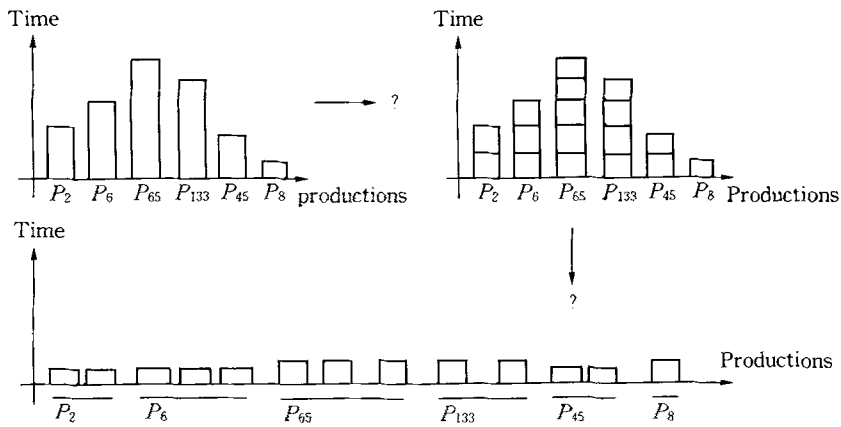

Fig. 3 Motivation for the parallel algorithm.

of computation to update the state. This, of course, is very desirable when the interpreter is implemented on a uniprocessor. However, this need not be the case with parallel computation. Such a conservative approach creates substantial variances in the processing times for the productions affected each cycle, thereby reducing the available production level parallelism as depicted in Fig. 2. Here for example productions P_{65} and P_{133} (processed together in some cycle) take much longer than the other productions affected, effectively reducing production

level parallelism.

Figure 3 presents the main motivation behind the algorithm DYNAMIC-JOIN. The basic idea is to split production state processing into much smaller units (as depicted by the middle graph) and then process these units in parallel if possible (as depicted in the lower graph). The approach presented here distributes *most* of the computation that is required "suddenly", over to more than one cycle and over to many processors. The state representation is substantially different from those used in RETE or TREAT. This representation allows the state information to be divided into independently processable units thereby enabling the exploitation of a second level of finer grain parallelism in addition to production level parallelism. Another feature of this representation is that it allows the maximum amount of computation (some of it potentially redundant) to be done on the state of a production. So in this respect, in the spectrum of possible state maintenance schemes the algorithm to be presented is almost at one extreme (Match Box[12] being more extreme) while TREAT is at the other (conservative) extreme with RETE and its variations explored by Gupta,[4] in between.

§4 A New Representation for State of a Production

Given a production P, with c condition,* a actions P: $C_1C_2...C_c \rightarrow A_1A_2...A_a$, let R_i denote the set of WMEs that satisfy the constant tests of C_i. A special working memory element, X—called the *null* WME—is assumed to satisfy all the condition elements of a production hence is in each R_i. Also associated with each condition element C_i is a set of $c-1$ *intercondition tests* T_{ij} ($j = 1, ..., c, j \neq i$). For working memory elements $\omega_i \in R_i$ and $\omega_j \in R_j$, $T_{ij}(\omega_i, \omega_j) =$ **true**, if the two WMEs are consistent with respect to the intercondition variable tests between condition elements i and j.**

At this stage, a simple production will be presented and examples will be developed on the way to clarify some of the concepts. The example production has three condition elements, and intercondition tests as follows:

```
(p example
  (type1 ^f1 1 ^f2 <x>)
  (type2 ^b1 <x> ^b2 <y>)
  (type2 ^b1 <y> ^b2 <> <x>)
  -->
  (make type1 ^f1 <x> ^f2 <y>)
```

1. $T_{12}(\omega_1, \omega_2)$: $T_{21}(\omega_2, \omega_1)$: $(\omega_1.f2 = \omega_2.b1)$
2. $T_{13}(\omega_1, \omega_3)$: $T_{31}(\omega_3, \omega_1)$: $(\omega_1.f2 \neq \omega_3.b2)$
3. $T_{23}(\omega_2, \omega_3)$: $T_{32}(\omega_3, \omega_2)$: $(\omega_2.b1 \neq \omega_3.b2)$ and $(\omega_2.b2 = \omega_3.b1)$

For example T_{23} captures the constraint that the b1 field value of WMEs matching the second condition element—bound to variable <x>—should be different from the b2 field value of WMEs matching the third condition element

* Some of which may be negative condition elements.

** One can use half the tests since $T_{ij}(\omega_i, \omega_j) = T_{ji}(\omega_j, \omega_i)$.

and that b2 field value of the WME matching the second condition element—bound to variable ⟨y⟩—should be equal to the b1 field value of the WME matching the third condition element.

Suppose that at some point during execution the working memory contains the following WMEs:*

W_1: (type1 ↑f1 1 ↑f2 12)
W_2: (type1 ↑f1 2 ↑f2 14)
W_3: (type2 ↑b1 12 ↑b2 14)
W_4: (type2 ↑b1 12 ↑b2 12)
W_5: (type2 ↑b1 14 ↑b2 34)

The sets R_i corresponding to the condition element memories would be:

$$R_1 = \{X, W_1\}$$
$$R_2 = R_3 = \{X, W_3, W_4, W_5\}.$$

For example, W_1 has a type field type1 and its f1 field has value 1 as required by the first condition of the example production.

The state of a production P, denoted by $S(P)$ is a *subset* of the cartesian product $R_1 \times R_2 \times \ldots \times R_c$ and consists of a set of *instance elements*. An instance element is a c-tuple of *slots*:

$$IE = \langle(t_1, \omega_1)\ (t_2, \omega_2)\ldots(t_c, \omega_c)\rangle.$$

The ith slot of the instance element consists of a *tag* t_i and a WME ω_i where $\omega_i \in R_i$. Within each instance element, any two WMEs $\omega_i \in R_i$ and $\omega_j \in R_j$ (either or both may be X) are consistent with respect to T_{ij} with the assumption that the null WME X satisfies all intercondition tests, that is $T_{ij}(\omega_i, X) = T_{ij}(X, \omega_j) = T_{ij}(X, X) =$ **true**. For example, the following are some of the instance elements that can be constructed for the example production with the state of the working memory as given above.

IE_1: $\langle(t_{11}, W_1)\ (t_{12}, W_3)\ (t_{13}, W_5)\rangle$
IE_2: $\langle(t_{21}, W_1)\ (t_{22}, W_4)\ (t_{23}, W_3)\rangle$
IE_3: $\langle(t_{31}, X)\ (t_{32}, X)\ (t_{33}, W_4)\rangle$
IE_4: $\langle(t_{41}, X)\ (t_{42}, W_5)\ (t_{43}, X)\rangle$
IE_5: $\langle(t_{51}, W_1)\ (t_{52}, X)\ (t_{53}, W_5)\rangle$
IE_6: $\langle(t_{61}, X)\ (t_{62}, W_4)\ (t_{63}, X)\rangle$
IE_7: $\langle(t_{71}, W_1)\ (t_{72}, X)\ (t_{73}, X)\rangle$

It can be seen that the definition for the instance elements above allows construction of instance elements that contain information that is redundant with respect to other instance elements. For example, in the set of instance elements above, IE_5 captures the information that W_1 in slot 1 agrees with W_5 in slot 3. However,

* In the following discussions W will be used to denote a specific WME and ω will be used to denote a generic element of a set of WMEs.

IE_1 contains the same information plus the information that W_3 in slot 2 agrees with both W_1 and W_5. Thus IE_5 is redundant. Similarly, the information in IE_6 is contained in the instance element IE_2. In order to formalize this notion of redundancy, the following relation between two instance elements of a production IE and IE′ is defined. An instance element IE is *covered* by IE′ (denoted as IE $\leq$ IE′) if for all i: $1 \leq i \leq c$, either $\omega_i = \omega_i'$ or $\omega_i = X$. This is equivalent to saying that IE′ contains the match information in IE. This relation is reflexive, antisymmetric and transitive and thus induces partial ordering on the set of instance elements. The instance elements that are not redundant are the maximal elements in this partial ordering.

The state of a production is defined to be the set of the instance elements that are not redundant. Thus in the example avove only IE_1, IE_2, IE_3 and IE_4 would make up the state. More formally, let

$$IES(P) = \{IE = \langle(t_1, \omega_1), ..., (t_c, \omega_c)\rangle: \omega_i \in R_i \wedge T_{ij}(\omega_i, \omega_j) = \textbf{true}, 1 \leq i < j \leq c\}$$

The state of a production, $S(P)$, can then be defined as:

$$S(P) = \{IE: IE \in IES(P) \wedge (\forall IE' \in IES(P) \wedge IE' \neq IE \wedge IE \not\leq IE')\}.$$

This way of keeping the state is *equivalent* to maintaining a "super" AND-node that joins all condition element memories dynamically (hence the name DYNAMIC-JOIN) whenever any joinable WMEs are added to any of them. The super AND-node concurrently maintains non-empty joins $R_{i_1} \otimes ... \otimes R_{i_j}$ for all $\{i_1, ..., i_j\} \subseteq \{1, ..., c\}$. Any such $R_{i_1} \otimes ... \otimes R_{i_j}$, $1 \leq j \leq c$, contains only those j-tuples of working memory elements $(\omega_{i_1}, ..., \omega_{i_j})$ that are all mutually compatible, and are *not* in any of the sets $\Pi_{i_1,...,i_j}(R_{k_1} \otimes ... \otimes R_{k_l})$ where $\{i_1, ..., i_j\} \subset$

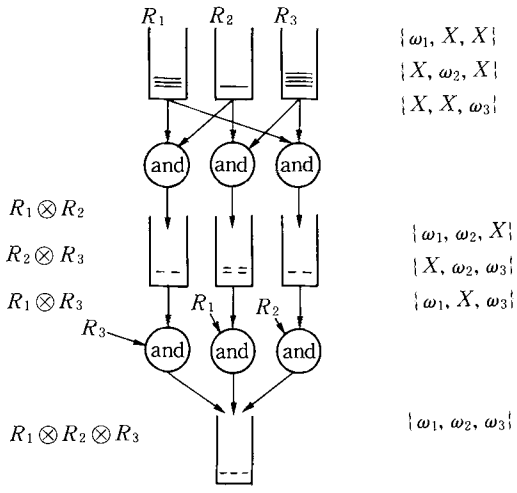

Fig. 4 State representation for the parallel algorithm.

$\{k_1, ..., k_l\}$ and $\Pi_{i_1,...,i_j}(...)$ denotes the relational projection operation[17] of the argument relation over columns $i_1, ..., i_j$.* For example the pair of mutually compatible WMEs (W_1, W_2) need not be kept in $R_1 \otimes R_2$, if there is a 3-tuple (W_1, W_2, W_3) in $R_1 \otimes R_2 \otimes R_3$, since the information in the former is available in the latter. Figure 4 presents a logical view of the state representation in terms of the intermediate state kept for production with 3 condition elements. A detailed example of how a state is formed and maintained, will be presented later in this paper to clarify some of the concepts above.

4.1 The Tags

A WME W matching condition element i of P (and hence in R_i) may be in the ith slot of one or more instance elements in $S(P)$. When instance elements are manipulated after insertions and deletions to and from the working memory, some redundant information may be generated. It is possible to detect and eliminate *some* of this redundancy by using additional information in each instance element. The tags associated with the slots in an instance element serve this purpose.

One of the occurrences of W in a given slot position is marked as containing the *master* copy of that working memory element for that slot position. Other occurrences are tagged as containing either *new* or *old* copies depending on how and when the working memory element is inserted into that slot. Instance elements that accommodate W during the cycle it is inserted to the working memory get to use the tag *new* or *master*. On the other hand, when new instance elements are generated from existing ones, some of the slots from the generating instance element are copied to the newly generated instance element. The slots whose WMEs are copied, are tagged with the tag *old*. In the upcoming discussions, the following symbols will be used to denote the tags.

(1) m: Indicates that the accompanying WME is master copy for this slot. There can be only one instance element with an m tag associated with a given working memory element in a given slot position.

(2) n: Indicates that the accompanying WME is a new copy for this slot. There may be more than one instance element that has an n tag for a WME filling a given slot position.

(3) o: Indicates that the accompanying WME is an old copy for this slot. This instance element was generated from another instance element and this slot's working memory element was copied from there.

(4) x: Indicates that the accompanying WME is the null element.

Given IE $= \langle(t_1, W_1)\ (t_2, W_2)...(t_c, W_c)\rangle$ and $M = \{s_1, s_2, ..., s_m\} \subseteq \{1, ..., c\}$, denoting a subset of the slots in the instance element, the following are interesting combinations of tags that convey useful information:

* The join relation can be abstracted as a collection tuples where each element of the tuple represents a WME.

(1) $m \geq 1$ and $t_{s_1} = t_{s_2} = \ldots = t_{s_m} =$ o: This instance element was generated from another instance element and the WMEs corresponding to the slots in M were copied from there. Hence, there is some other instance element in the state with the same WMEs in these slots with at least one of the corresponding tags being n or m.

(2) $m = 1$ and $t_{s_1} =$ n: This indicates that there is some other instance element in the state that has the m tag in the slot s_1.

In the state of a production, all the instance elements have at least one slot that has an m or n tag. Otherwise, if all slots in an instance element have o or x tags, then instance must have been copied from another one with exactly the same working memory elements in the slots with the o tags, hence the former instance element would be redundant. Also, if an instance element has only one slot full, then this slot should have the m tag, and there should not be any other instance element containing the same WME in the same slot position.

§5 Processing an Instance Element

5.1 Matching a New Working Memory Element to an Instance Element

Matching a new WME W_{new} satisfying the ith condition element of the production, to an instance element $\mathsf{IE} = \langle(t_1, \omega_1)\ (t_2, \omega_2)\ldots(t_c, \omega_c)\rangle$ involves evaluating the intercondition tests $T_{ij}(\mathsf{W}_{\text{new}}, \omega_j)$ for all j such that $i \neq j$, $\omega_j \in R_j$ and $\omega_j \neq \mathsf{X}$. Let F be the number of slots in the instance element with a non-null working memory element, and let M denote the set of indices of such slots with working memory elements agreeing with the new WME, that is

$$M = \{j: i \neq j \wedge \omega_j \in R_j \wedge \omega_j \neq \mathsf{X} \wedge T_{ij}(\mathsf{W}_{\text{new}}, \omega_j) = \textbf{true}\}.$$

The following may be the *only* outcomes of the matching W_{new} to an instance element:

(1) $|M| = 0$: The new WME is *not* consistent with any of the non-null working memory elements in the slots of the instance element, or all slots other than the ith have null WMEs. No new match information can be derived from instance element.

(2) $|M| = F$: In this case, the ith slot of the instance element IE contains X, and all the intercondition tests evaluate to **true**. The update to the state consists of modifying this instance element to

$$\mathsf{IE} = \langle(t_1, \omega_1)\ldots(t_{i-1}, \omega_{i-1})\ (\mathsf{m/n}, \mathsf{W}_{\text{new}})\ (t_{i+1}, \omega_{i+1})\ldots(t_c, \omega_c)\rangle$$

that is, the null WME in slot i is replaced with the new WME along with a n or m tag.

(3) $1 \leq |M| < F$: This is the case where a new instance element *may* be generated. There are two subcases that may lead to this case:

(a) All the tested slots of the instance element agree with W_{new} but $\omega_i \neq X$. This case is analogous to the case (2) above, except that the ith slot is *not* empty and hence can not accommodate the matching working memory element.

(b) Some of the WMEs in the tested slots of the instance element disagree with W_{new}.

If M satisfies any of the conditions presented at the end of Section 4.1, then clearly any new instance element that will be generated from this one will be redundant since the same information will be generated from some other instance element. In this case, no new instance element needs to be generated. Otherwise, a new instance element

$$IE' = \langle (t'_1, \omega'_1)...(t'_{i-1}, \omega'_{i-1})\ (t'_i, \omega'_i)\ (t'_{i+1}, \omega'_{i+1})...(t'_c, \omega'_c)\rangle$$

is generated to be added to the state, where

- $\omega'_j = \omega_j$ and $t'_j = o$ for all $j \in M$ (copy all agreeing slots with tags set to old.)
- $\omega'_j = X$ and $t'_j = x$ for all j, $j \neq i$ and $j \notin M$ (put null working memory elements to slots corresponding to disagreeing or null slots in the original instance element.)
- $\omega'_i = W_{new}$ and $t'_i = n$ or m (and insert the new WME to the ith slot.)

5.2 Updating an Instance Element after Deleting a Working Memory Element

Processing the effect of the deletion of a WME W_{del} on an instance element of a production is much easier since no intercondition variable consistency tests have to be performed. Basically, all the slots of the instance element that contain W_{del} are modified so that now contain the null WME X. The modified instance element *may* contain information that is now redundant. In this case, the instance element is deleted from the state.

§6 Parallel Algorithms for Maintaining the State of a Production

The condition element memories of a production (the sets R_i) change as WMEs matching its condition elements are inserted and deleted. In the state representation above, each instance element in state of a production has to be processed in response to any change to the condition element memories associated with the production. State processing involves two phases:

(1) processing the instance elements for matches to the WME inserted or deleted,

(2) checking the instance elements for redundancy and eliminating any redundant instance elements

In the first phase individual instance elements can be processed independently in parallel provided sufficient number of processors are available. Thus a second level of parallelism in state processing can be exploited in addition to the production level parallelism exhibited by the production system.

In the second phase, *some* of the instance elements that are modified and/or generated during first phase are eliminated if they are found to be redundant. It should be noted that any redundancy checks in this phase involves checking those instance elements against others, since their redundancy can not be detected by using tags only. Whenever some instance element is found to be redundant, any critical tag information in that instance element has to be transferred to the instance element that covers it. This guarantees that m and n tags in any instance element do not get lost.

The following discussions will present algorithms for parallel processing of the set of instance elements of a production. The model of parallel computation that will assumed for these parallel algorithms (shown in Fig. 5) has the following properties:

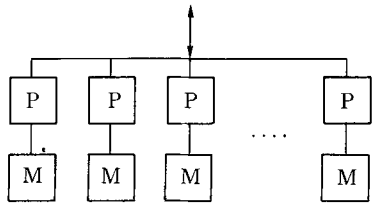

Fig. 5 Logical view of the parallel model of computation.

- The state of a production is assigned to be processed by $k \geq 1$ processors. Each processor holds a subset of the instance elements. Ideally there would be only one in each processor.*
- Each processor has its own local memory and processors do not share memory.
- The processors receive working memory changes over a communication medium and also communicate with each other (during redundancy checking) using this medium.

6.1 Processing the State after an Insertion

When a WME is inserted to the working memory and matches one or more condition elements of a production, all the instance elements of that production have to be processed in order to determine what additional information should be added to the state. However in order to strictly maintain the non-redundancy constraint on the state, any redundant instance elements should

* But, as explained later, a processor may be shared by more than one production.

be eliminated after processing the state for each condition element matching the WME. The reasons for enforcing the non-redundancy constraint is to prevent spurious growth in the state and in certain instances prevent the generation of incorrect information in the state of productions with negative condition elements.*

When the state of a production is processed for an insertion after a working memory element matches one of the condition elements, there will be three sets of instance elements as shown in Fig. 6.

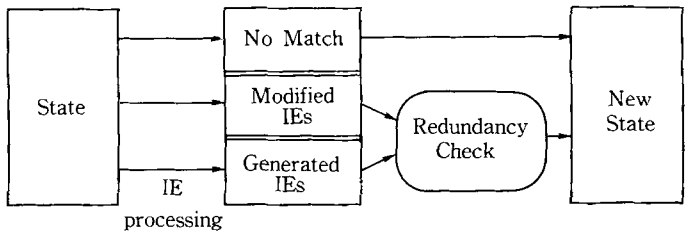

Fig. 6 Changes in the state after an insertion.

(1) instance elements that could accommodate the new WME and hence were *modified*,
(2) instance elements that were *generated* from other instance elements that could not accommodate the new WME, and
(3) instance elements that *did not match* the newly inserted working memory element.

If the state did not have any redundant instance elements prior to the insertion, it can be seen that none of the modified instance elements can be redundant after the insertion. The reason for this is that since these instance elements were not redundant before the insertion, modifying one of their empty slots with the new WME can not make them redundant. This implies that the redundant instance elements (if any) are among the set of generated instance elements, and they will be covered by either a modified instance element or by some other generated instance element.

The only issue that seems to require some form of communication in the instance element processing phase of state processing after an insert involves resolving which instance element in the state gets to use the master tag for the inserted working element in the matching condition element slot and this can be solved by a number of simple approaches.[11]

Once the selection is over, all the processors start processing their subset of the instance elements in parallel. When all processors processing the instance

* However, this constraint can be relaxed somewhat provide certain conditions are met.[11]

elements are finished, the redundancy checking phase starts. In order to eliminate any such instance elements, the following sequence of operations take place:

(1) Starting with processor 0, processors with generated instance elements broadcast these to all the processors assigned to that production.
(2) After an instance element is broadcast for redundancy checking, all the processors check in parallel to see if any of their modified or generated instance elements cover the broadcast instance element. If the instance element is covered, and it has the m tag, then this tag is copied to the covering instance element. Otherwise, the processor with the smallest number of instance elements for that production, adds the broadcast instance element to its subset of the state. This tries to attain a balanced distribution of instance elements across the processors assigned to a production.
(3) When all processors broadcast their potentially redundant instance elements, the redundancy check phase is over, and the processors either complete processing this production or proceed with the next condition element matching the inserted WME.

6.2 Processing the State after a Deletion

When a WME is deleted from the working memory, all the instance elements in the state of a production have to be processed in order to modify those that contain the deleted WME. Contrary to the case in insertion, the state has to be processed only once even if the WME had matched more than one condition element of the production. Again, processing the state consists of two phases: processing the instance elements for the deletion, and performing the redundancy check afterwards to eliminate any redundant information that could not be detected by the use of tags.

After the state is processed for a deletion, there will be two sets of instance elements:

(1) instance elements that were modified (had one or more slots replaced with X) during the delete,
(2) instance elements that were not modified during the delete.

The only instance elements that may be potentially redundant after a delete are among the first set of instance elements.

As in the insert operation, the instance elements are assumed to be on k processors. Once the selection is over, all the processors start processing their subset of instance elements modify any instance elements that contain the delete WME in their slots.

§7 A Short Example

This section will present a short example to demonstrate the concepts presented above—a more complete examples can be found in author's thesis.[11] The example will use the production presented earlier in Section 4.

- **Cycle 1**: **INSERT** W_1 = (type1 ↑f1 1 ↑f2 12)

The working memory element matches the first condition element. Initially there are no instance elements in the state, so processor 0 generates a singleton instance element with only the first slot filled.

Proc #	State	Generated IES
0	— —	⟨(m, W_1) (x, X) (x, X)⟩

The generated instance element is then added to the state.

Proc #	State
0	⟨(m, W_1) (x, X) (x, X)⟩

- **Cycle 2**: **INSERT** W_3 = (type2 ↑b1 12 ↑b2 14)

This WME matches both the second and the third condition elements. First the state is processed for the second condition element. In instance element IE_1, W_3 agrees with W_1 in slot 1. Hence W_3 fills up the empty second slot in this instance element modifying it to:

Proc #	State
0	IE_1: ⟨(m, W_1) (m, W_3) (x, X)⟩

Since there are no generated instance elements, there is no need for a redundancy check. Now the resulting state is processed for the match to the third condition element. In IE_1, W_1 in slot 1 agrees with W_3 in slot 3, however W_3 in slot 2 disagrees with W_3 in slot 3 since the variable ⟨y⟩ can not be bound to the same value in both WMEs. In this case, a new instance element is generated with the information in the matching slot copied.

Proc #	State	Generated IES
0	IE_1: ⟨(m, W_1) (m, W_3) (x, X)⟩	⟨(o, W_1) (x, X) (m, W_3)⟩

After redundancy check, the generated instance element is added to the state on processor 1. The state of the production is now

Porc #	State
0	IE_1: ⟨(m, W_1) (m, W_3) (x, X)⟩
1	IE_2: ⟨(o, W_1) (x, X) (m, W_3)⟩

§8 A Parallel Processor Architecture

The following sections present the architecture for a parallel processor for production systems that can be used to implement the parallel algorithm

presented in the preceding sections followed by a high-level description of its operation. Subsequently, results from simulation experiments with the proposed algorithm and architecture will be provided.

§9 Structure of the Parallel Processor

The parallel processor is envisioned as back-end match engine (as depicted in Fig. 7) that implements the match part of the production system interpretation cycle. This engine is connected to a front-end system that is responsible for interfacing to the users, executing the act and conflict resolution functions in addition to controlling the parallel processor. The interface between the match engine and the front-end controller is a simple one: for every action that modifies the working memory, a command (insert or delete) is sent to the match engine along with the WME involved. The match engine responds by sending any changes to the contents of the conflict set. It is expected that the front end system will be a conventional processor that is fast enough to match the performance of the back end.

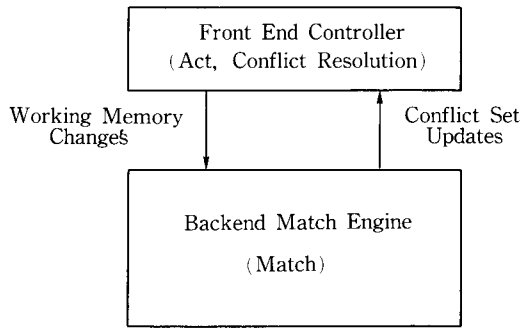

Fig. 7 Partitioning of PS Interpreter functions.

The parallel algorithm presented earlier exploits two levels of parallelism: processing the state of the productions affected by a working memory chage, in parallel, and processing the instance elements in the state of each production in parallel. This necessitates a structure with a large number of processors—possibly in the range of 256 to 1024. However, contrary to other proposed highly-parallel architectures for production systems (e.g., DADO,[15] NON-VON[7]), instead of allocating dedicated processors to each production, the processors are to be multiplexed to the large number productions in the production system, since during each cycle the states of only a small number of productions need to be processed. The processors have their own memories, do not share any memory and operate in MIMD mode (see Ref. 11) for a detailed discussion on the requirements of the architecture). These considerations above necessitate a structure where groups of processors can be organized into groups of possibly different sizes and that the organization of these clusters can change

from cycle to cycle depending on the productions that are being processed during the cycle. The communication network provides dedicated paths among the processors organized into a group.

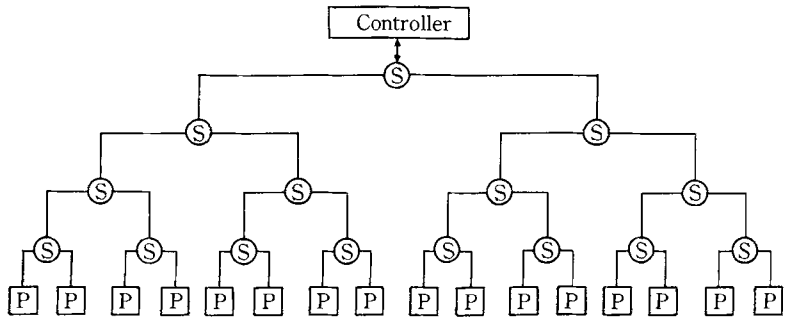

Fig. 8 Structure of the proposed parallel processing system.

A very suitable structure that fits the requirements duscussed above is an array of $K = 2^M$ processors at the leaves of a binary tree consisting of $K - 1$ switches as depicted in Fig. 8. The switches constitute the communication network that connects the processors to the front-end controller and to each other. This parallel processor structure is modular and extensible and can be implemented employing switches and processors built with the VLSI technology. Groups of processors (of certain selected sizes) can be created on this structure by allowing certain switches logically disconnect subtree underneath them from the rest of the tree. Each of these groups then have their own independent communication networks during redundancy checking. Furthermore the configuration and the number of the groups can be changed from cycle to cycle, though within the limits of the tree organization. The binary tree of the switches also provides a very suitable organization for providing certain functions during redundancy checking.

This parallel processor organization is substantially different from those of DADO[15] and NON-VON,[7] since processors are only at the leaves, and the internal nodes of the tree are used to implement a communication and reconfiguration network. Furthermore, in contrast to NON-VON in which processing elements are very simple pattern matching machines that operate with instructions received from a control unit, the processors in this organization have the complexity of a microprocessor and operate in MIMD mode.

The two levels of parallelism mentioned earlier can be mapped onto this structure as follows: Productions are assigned to be processed by a processors that comprise the leaves of a complete subtree of switches. A production P is assigned to be processed by $k = 2^m$ processors that correspond to the leaves of

Table 1 Hypothetical processor requirements of productions for an example production system.

Productions :	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Processors :	2	2	4	8	4	4	2	1	1	4	8	8

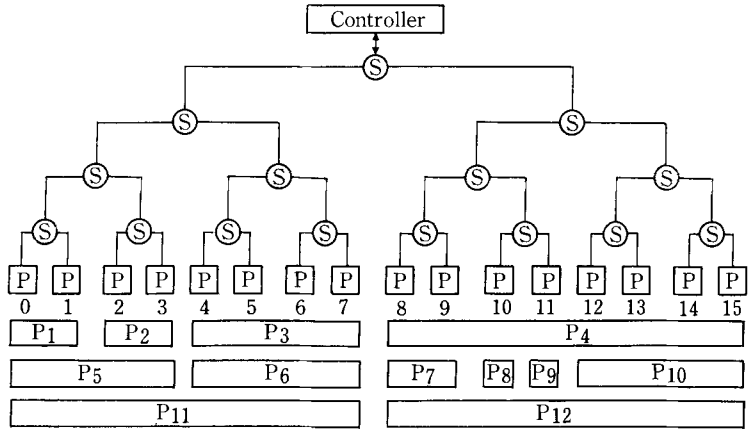

Fig. 9 A possible assignment of productions for the example production system.

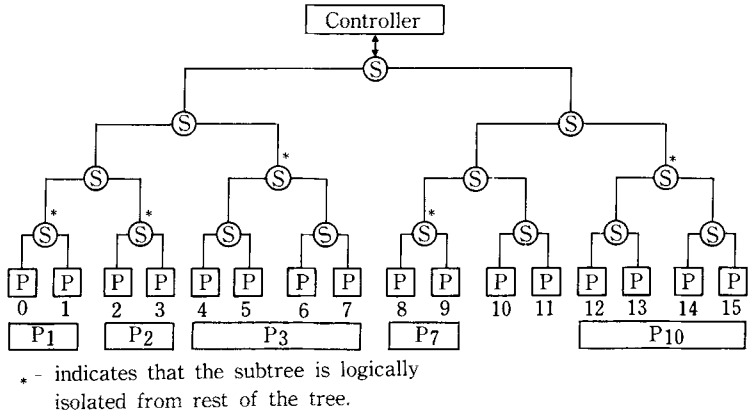

Fig. 10 Switch configuration for processing P1, P2, P3, P7 and P10.

a proper subtree of the large tree. Thus, P can only be assigned to the leaves of one of the 2^{M-m} subtrees, with the leftmost processor labeled 0, 2^m, $2 \cdot 2^m$, ..., $(2^{M-m} - 1) \cdot 2^m$. The following examples will clarify how the switches operate to implement reconfiguration of the processors.

Suppose an example production system has 12 productions, the parallel processor has $K = 16$ processors, and the productions have processor requirements as shown in Table 1. Figure 9 presents one way of assigning these productions to the processors. For example, if during a cycle, productions P1, P2, P3, P7 and P10 are to be processed together, then it will be possible to do this

without any processor having to process more than one production. In this case, the processors would be configured into an organization where five groups of processors would operate independently as depicted in Fig. 10. The switches at the roots of the subtrees to which these productions are assigned would (logically) disconnect the subtrees from the rest of the tree so that the redundancy check phases of these productions can proceed concurrently using the independent communication paths established via the reconfiguration. If on the other hand P1, P2, P4, P5, P7, P12 have to be processed as a result of an action during a cycle, then processors 0 through 3 and 10 through 15 would sequentially process the state of two productions, while processors 8 and 9 would process the states of three productions.

§10 Operation of the Parallel Processor

At a very high level, the interface of the parallel processor to the front end controller is a simple one. The controller sends commands of the form

- INSERT(W)
- DELETE(W)

to insert or delete WMEs from the working memory, and the processors respond by sending messages to insert or delete production instantiations from the conflict set maintained by the front-end controller.

When the controller issues an insert command—INSERT(W)—the field values for the WME are broadcast to all the processors via the tree of switches.

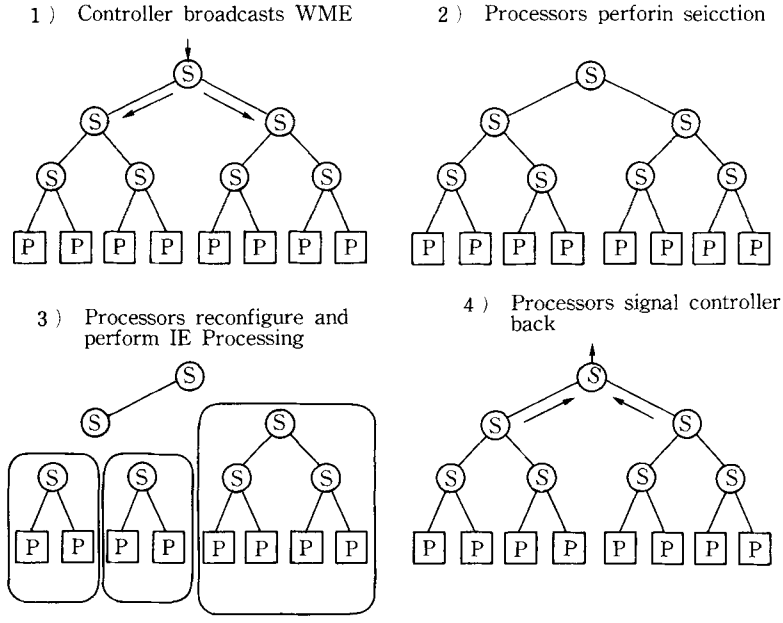

Fig. 11 Operation of the parallel processor during a match cycle.

Each processor at the leaves of the tree perform the selection phase of the match cycle for the subset of the productions assigned to it. This selection phase within each processor can be implemented by very small versions of the discrimination network used in the uniprocessor RETE interpreter.[3] The selection process within each processor identifies a set of productions (along with their matching condition elements) whose states have to be processed by that processor.

After the selection, the instance elements of those productions selected are processed as described earlier (see Ref. 11) for the details of the procedures). When a processor completes processing the states of the productions assigned to it, it sends a DONE signal to its parent switch. When a switch receives the DONE signal from both of its sons (processors or switches), it passes the signal to its parent switch. Finally when the topmost switch receives the signal from its sons, it passes the signal to the controller indicating that the match cycle has been completed. The operation for a delete operation is exactly the same. Figure 11 outlines the phases of a typical match cycle. Redundancy check stage follows the instance element processing during which the processors are reconfigured to form groups of differents sizes and each groups proceeds with its own redundancy check.

§11 Simulation of the Parallel Algorithm and Architecture

This section describes the results from a limited set of simulated executions of a number of production systems with the parallel algorithm presented earlier in order to observe its behavior and to get a feeling of potential performance. These simulated executions have been implemented with a simulator that has been built on top of the Lisp-based RETE interpreter running on a VAX 11/780 (see Ref. 11) for a detailed discussion on various architectural and timing assumptions made).

The simulator was used with four production systems: XSEL, R1, MUD and EP-SOAR with 1303, 2153, 872 and 62 productions respectively. Traces from two application runs of XSEL and one run of the others were available. The first three systems are relatively large real application systems while EP-SOAR is a very prototype small system. However EP-SOAR has a number of characteristics that make it interesting for DYNAMIC-JOIN. On the average each production of EP-SOAR has 10 condition elements—2 to 3 times the number of condition elements of the other ones. With the exception of 8 productions (with 58, 47, 27, 20, 20, 16, 13, and 10 condition elements) the productions in EP-SOAR behave reasonably. There are however a number of reasons why systems like EP-SOAR are not very suited for this parallel algorithm.

11.1 Results from Simulations

(1) Distribution of average number of instance elements of productions

The behavior of the state of the production can be characterized by the

Table 2 Distribution of average number of instance elements in the states of productions of XSEL.

XSEL Run 1				XSEL Run 2			
RANGE	P's	%	Cum. %	RANGE	P's	%	Cum. %
0	579	43.0%	43.0%	0	591	43.9%	43.9%
0-1	332	24.7%	67.7%	0-1	322	23.9%	67.8%
1-2	129	9.6%	77.3%	1-2	138	10.3%	78.1%
2-3	63	4.7%	82.0%	2-3	73	5.4%	83.5%
3-4	46	3.4%	85.4%	3-4	34	2.5%	86.0%
4-5	29	2.2%	87.6%	4-5	27	2.0%	88.0%
5-6	28	2.2%	89.8%	5-6	30	2.2%	90.2%
6-7	15	1.1%	90.9%	6-7	13	1.0%	91.2%
7-8	3	0.2%	91.1%	7-8	21	1.6%	92.8%
8-9	6	0.4%	91.5%	8-9	2	0.1%	92.9%
9-10	4	0.3%	91.8%	9-10	5	0.3%	93.2%
10-15	41	3.1%	94.9%	10-15	28	2.3%	95.5%
15-20	35	2.6%	97.5%	15-20	34	2.5%	98.0%
20-30	21	1.6%	99.1%	20-30	20	1.5%	99.5%
30-50	12	0.8%	99.9%	30-50	7	0.5%	100.0%
50-100	2	0.1%	100.0%	50-100	0	0.0%	100.0%

average number of instance elements that have to be processed during every cycle that the production is affected. Tables 2 and 3 present the distribution of the average number of instance elements for the productions of the systems considered.*

From these results it can be seen that for the two large systems XSEL and R1, more than 90% of the productions have less than 10 instance elements on the average during execution. For MUD around 80% of the productions have on the average less than 10 instance elements during executions. For EP-SOAR the cumulative percentage for the 0-10 range is smaller since the number of productions is very small compared to the other systems.**

* Productions that have been split up have been counted as distinct productions.[11] In XSEL 42 productions (3.2% of the productions) needed to be split up while in R1 only 22 productions (1.0% of the productions) needed to be split up. In MUD, 40 productions that were sensitive to every change to the working memory were removed from the system. These were productions for tracing and debugging and their inclusion in the system would distort the statistics and increase simulation time unnecessarily. An additional 43 productions in MUD (4.9% of the productions) were split up. In EP-SOAR, 8 productions with very large number of condition elements (with 58, 47, 27, 20, 20, 16, 13, and 10 condition elements) were removed from the system, since they could not be handled even after splitting. A total of 16 productions (25% of the productions) with relatively large number condition elements (8 to 19) were split up.

** The average number of WMEs in the working memory during each cycle of the XSEL runs were 177 and 181, while the corresponding figures for the R1 run were 243. The MUD run had on the average 235 WMEs, and EP-SOAR had on the average 180 WMEs. These numbers indicate that the working memory size during these runs is relatively small which may be one of the reasons for the distributions above. Selective productions will be satisfied with a very small subset of these working memory elements and thus their state will have a small number of instance elements.

Table 3 Distribution of average number of instance elements in the states of productions of R1, MUD and EP-SOAR.

R1				MUD				EP-SOAR			
RANGE	P's	%	Cum. %	RANGE	P's	%	Cum. %	RANGE	P's	%	Cum. %
0	1650	75.8	75.8	0	541	62.0	62.0	0	6	7.0	7.0
0-1	139	6.4	82.2	0-1	56	6.4	68.4	0-1	1	1.2	8.2
1-2	175	8.0	90.2	1-2	37	4.2	72.6	1-2	1	1.2	9.4
2-3	19	0.9	91.1	2-3	12	1.4	74.0	2-3	7	8.1	17.5
3-4	49	2.3	93.4	3-4	8	0.9	74.9	3-4	6	7.0	24.5
4-5	12	0.6	94.0	4-5	14	1.6	76.5	4-5	5	5.8	30.3
5-6	10	0.5	94.5	5-6	2	0.2	76.7	5-6	2	2.3	32.6
6-7	15	0.7	95.2	6-7	15	1.7	78.4	6-7	5	5.8	38.4
7-8	6	0.3	95.5	7-8	7	0.8	79.2	7-8	4	4.7	43.1
8-9	1	0.04	95.5	8-9	10	1.1	80.3	8-9	4	4.7	47.8
9-10	1	0.04	95.6	9-10	5	0.6	80.9	9-10	2	2.4	50.2
10-15	18	0.8	96.4	10-15	15	1.7	82.6	10-15	13	14.6	64.8
15-20	27	1.2	97.6	15-20	6	0.7	83.3	15-20	5	5.8	70.6
20-30	18	0.8	98.4	20-30	42	4.8	88.1	20-30	5	5.8	76.4
30-50	25	1.1	99.5	30-50	97	11.0	99.1	30-50	7	8.2	84.6
50-100	10	0.5	100.0	50-100	4	0.5	99.6	50-100	11	12.9	97.5
				100-200	3	0.4	100.0	100-200	2	2.5	100.0

Table 4 Statistics on the behavior of the algorithm for four production systems.

	XSEL Run1	XSEL Run2	R1 Run	MUD Run	EP-SOAR Run
Number of WM Actions	1868	1945	1665	2074	924
Avg. Prods. Processed/Cycle	23.9	23.9	14.6	24.4	9.5
Avg. IEs Processed/Cycle	255.7	267.0	206.5	316.2	222.3
Avg. IEs Generated/Insert	24.6	25.6	25.6	37.5	122.1
Avg. IEs Modified/Insert	20.0	20.3	15.1	29.6	22.7
Avg. IEs Modified/Delete	47.2	44.8	39.0	53.6	71.7
Avg. Redundant IEs eliminated with tags/Delete	21.4	22.6	17.7	33.0	24.4
Avg. IEs Broadcast for Redundancy Checking/Insert	22.8	24.1	25.7	23.0	122.0
Avg. Redundant IEs eliminated with Redundancy Checking/Insert	4.8	4.6	10.2	4.2	104.9

[2] Run time statistics for the parallel algorithm

The set of figures presented in Table 4 provide information about the aggregate behavior of the production system as it is being interpreted with the DYNAMIC-JOIN.* The ratio of instance elements processed per cycle to processors used per cycle is an approximate measure of this utilization. For the systems considered, it is 0.82 and 0.84 for XSEL runs, 0.73 for R1 to 0.93 for EP-SOAR runs; that is on the average a small number of processors are essen-

* It should be noted that some of these nunbers are averages over productions and cycles. For instance, only one production may be responsible for half of the new instance elements generated during a cycle. It is very hard to present such information without voluminous per production statistics.

tially idle.

[3] Timing results

The time for a match cycle is the sum of the selection time and the state processing time. Since our algorithm is only concerned with the state processing part of the match cycle, the timing figures presented in Table 5 correspond to the average time the state processing phase of the match cycle takes on a simulated parallel execution of these systems with certain assumptions.[11] To these numbers one would have to add the time for selection that takes place within each processor. These numbers are estimated to be around 42 microseconds for selection for XSEL and R1, 80 microseconds for MUD and 38 microseconds for EP-SOAR (see Ref. 11) for details of how this time is determined). Assuming these selection times, a match cycle would complete in about 140 microseconds for XSEL, in 205 microseconds for R1, in 153 microseconds for MUD, and 440 microseconds for EP-SOAR. These translate to $\approx$7000 wme-actions/sec for XSEL, 4900 wme-actions/sec for R1, 6500 wme-actions/sec for MUD, and 2200 wme-actions/sec for EP-SOAR. They can be favorably compared with the 5 to 10 milliseconds—100 to 200 wme-actions/sec—for the VAX 11/780 BLISS-based interpreter. They represent an order of magnitude improvement over a VAX 11/780 even if we assume that the uniprocessor interpreter is highly optimized and can perform 500 to 1000 wme-actions/sec. Similarly they compare favorably to about 550 microseconds/wme-action reported for NON-VON[7] with 16K simple processing elements and to the estimated 5 milliseconds/wme-action for 1023 processor DADO2 using TREAT.[14,8] In terms of overall complexity, the architecture proposed here is simpler than both DADO and NON-VON in the sense that only the leaves have processors, and for the production systems considered in this paper, DYNAMIC-JOIN requires less number of processors than both the 16K processor NON-VON and 1023 processor DADO and achieves a better (simulated) performance than both. On the other hand, on shared memory multiprocessors Gupta's parallel RETE algorithm achieves very good performance (11,250 wme-changes/sec with 64-processors[4] but this algorithm relies on an associative hardware scheduler and performance is considerably lower with distributed software schedulers. In terms

Table 5 Average state processing time from simulated executions.

	XSEL Run1	XSEL Run2	R1 Run	MUD Run	EP-SOAR Run
Avg. Number of Processors used/Cycle	308	315	284	390	239
Std. Deviation	395	401	348	426	346
Avg. State Processing Time/Cycle (μsecs)	97	101	163	73	402
Ste. Deviation	125	130	292	115	710
Avg. IE Processing Time/Cycle (μsecs)	62	64	55	41	93
Std. Deviation	56	58	51	43	126
Avg. Red. Check Time/Cycle (μsecs)	35	37	108	32	309
Std. Deviation	90	94	274	75	425

of complexity, such shared memory systems are definitely more complex as they have to provide non-trivial cache systems, and complicated memory-processor interconnections.

§12 Conclusions

We have presented a parallel production system interpretation algorithm called DYNAMIC-JOIN along with a parallel processor architecture that is at one extreme of the spectrum of possible ways of keeping production state. The main motivation was to have a state representation for productions that would allow the components of the state be processed in parallel. Although such a state representation is successful in dividing up the state into parallel processable units, it presents an additional difficulty since an inherently sequential phase of redundancy checking has to be considered. This somewhat hampers the performance improvements that can be obtained by using such a state representation. At the extreme for example, it can be seen that in the case of EP-SOAR where the average production has about 10 condition elements, the redundancy check time essentially dominates the predicted cycle time.

Although the set of runs considered is rather limited, the production systems considered are not toy systems, but are real large application systems. The most important result of this research is the observation that, contrary to initial intuitions, the states of the productions do not grow outrageously. *In fact, such a behavior seems to be the exception rather than the rule.* It is certainly possible to write production systems where most or all of the productions behave adversely. This algorithm would probably not be appropriate for these. It would also be inappropriate for production systems that maintain large working memories. The systems considered here had average working memory sizes in the few hundreds and this factor has certainly helped. The more important factor that has been helpful, is that in general productions are sensitive to very small subsets of the working memory, and the intercondition tests are restrictive enough to prevent the cross-products from growing large.

The predicted performance results discussed in the preceding section are better than the results reported for other massively parallel architectures suggested for production systems and improvements in distributed redundancy checking would improve the performance of our machine significantly.

DYNAMIC-JOIN can also be adapted to a hypercube architecture (e.g., a low communication overhead system like the Intel iPSC/860).* The different (powers of 2) size subtrees in our architecture would map to cubes of similar dimension in a hypercube architecture. Different cubes could concurrently process the instance elements of the productions assigned to them.

There are a few comments that can be made about the negative parts of the proposed algorithm. It definitely is not suitable for production systems with

* Suggested by one of the referees.

productions having large number of condition elements. There are three reasons for this: Large number of condition elements mean a correspondingly large number of intercondition tests which increase memory requirements for a productions. The large number of condition elements increase the time it takes to process an instance element of the state. The most important is the observation that productions with a large number of condition elements tend to generate a large number of instance elements and these have to be dealt during redundancy checking phase. Thus for example EP-SOAR is a production system that would not be suitable for this parallel algorithm. Although the states of productions in EP-SOAR do not grow very large (excepting the ones with the very large number of condition elements) a large number of instance elements are generated after every insert which are subsequently deleted during redundancy checking.

Table 6 Improvements in production level parallelism.

	XSEL Run1	XSEL Run2	R1 Run
Average number of productions processed in each cycle	23.79	23.76	14.38
Production level parallelism with parallel RETE	4.11	4.04	6.13
Production level parallelism with our algorithm	14.26	14.26	7.73
Production level parallelism with DYNAMIC-JOIN considering only instance element proc. time	19.70	20.05	12.12

An interesting question to ask is whether the proposed algorithm was able improve the production level parallelism by smoothing out the variances state processing time across productions processed in each cycle. The answer from these set of runs is a reserved yes. Table 6 from Ref. 11) presents the improvements in production level parallelism in the XSEL and R1 runs. It can be seen that in the case of XSEL there is about a three fold improvement in production level parallelism indicating that most the variances have been smoothed out. However, the same is not true for R1 since the redundancy check time for that run is significantly larger. On the other hand if only the instance element processing times are considered, it can be seen that there is dramatic increase in the production level parallelism. Any algorithmic and hardware-based improvements for redundancy checking would substantially improve production level parallelism and the performance of the model presented in this paper.

Acknowledgements

I would like to thank my advisor H. T. Kung for his support during this work. I would also like to thank Roberto Bisiani, Charles Forgy and Allen

Newell, the other members of my committee, for their comments. Anoop Gupta provided a lot of ideas and engaged in discussions in all aspects of parallel processing of production systems. Larry Rudolph also provided some guidance during the initial conceptualization of the model and algorithm presented here. The anonymous referees also provided valuable comments about the contents and the presentation of this paper.

References

1) Forgy, C. L., "On the Efficient Implementation of Production Systems," *Ph. D. thesis*, Department of Computer Science, Carnegie Mellon University, February 1979.
2) Forgy, C. L., *OPS5 User's Manual*, Department of Computer Science, Carnegie Mellon University, 1981.
3) Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Match Problem," *Artificial Intelligence*, *19*, *1*, September 1982.
4) Gupta, A., "Parallelism in Production Systems," *Ph. D. thesis*, Department of Computer Science, Carnegie Mellon University, 1985.
5) Gupta, A., Forgy, C. L., Kalp, D., Newell, A. and Tambe, M., "Results of Parallel Implementation of OPS5 on the Encore Multiprocessor, in *Proceedings of International Conference on Parallel Processing*, IEEE Computer Society, pp. I271-280, 1988.
6) Gupta, A., Forgy, C. L. and Newell, A., "High-Speed Implementations of Rule-Based Systems," *ACM Transactions on Computer Systems*, *7*, *2*, May 1989.
7) Hillyer, B. K. and Shaw, D. E., Execution of OPS5 Production Systems on a Massively Parallel Machine," *Technical Report*, Department of Computer Science, Columbia University, September 1984.
8) Miranker, D. P., "Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE," in *Proceedings of International Conference on Fifth Generation Computer Systems*, 1984.
9) Miranker, D. P., *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Pittman/Morgan-Kaufman, Los Altos, CA, 1989.
10) Miranker, D. P. and Lofaso, B. J., "The Organization and Performance of a TREAT-Based Production System Compiler," *IEEE Transactions on Knowledge and Data Engineering*, *3*, *1*, pp. 3-10, March 1991.
11) Oflazer, K., "Partitioning in Parallel Processing of Production Systems," *Ph. D. thesis*, Department of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1987.
12) Perlin, M. W., "The Match Box Algorithm for Parallel Production System Match," *Technical Report*, *CMU-CS-89-163*, School of Computer Science, Carnegie Mellon University, 1989.
13) Schreiner, F. and Zimmerman, G., "PESA I—A Parallel Architecture for Production Systems," in *Proceedings of the 1987 International Conference on Parallel Processing* (S. K. Sahni, ed.), pp. 166-169, August 1987.
14) Stolfo, S. J., "Initial Performance of the DADO2 Prototype," *IEEE Computer*, *20*, *1*, pp. 75-84, January 1987.
15) Stolfo, S. J. and Miranker, D. P., "DADO: A Parallel Processor for Expert Systems," in *Proceedings of the International Conference on Parallel Processing*, IEEE, pp. 74-82, 1984.
16) Stolfo, S. J. and Miranker, D. P., "The DADO Production System Machine," *Journal of Parallel Distributed Computation*, *3*, pp. 269-296, June 1986.
17) Ullman, J., *Principles of Database Systems*, Computer Science Press, 1987.